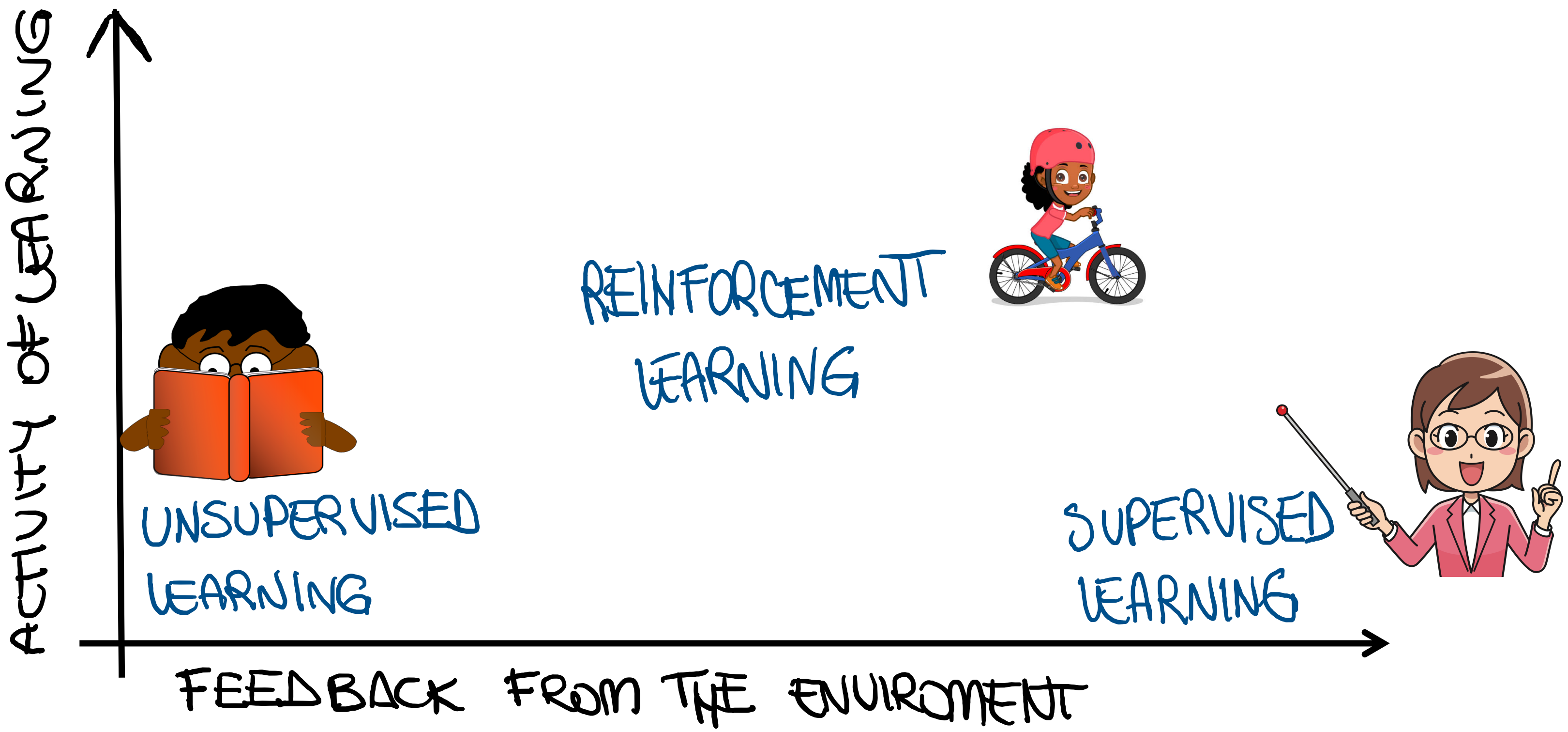
Cyber-Physical Systems

Laura Nenzi

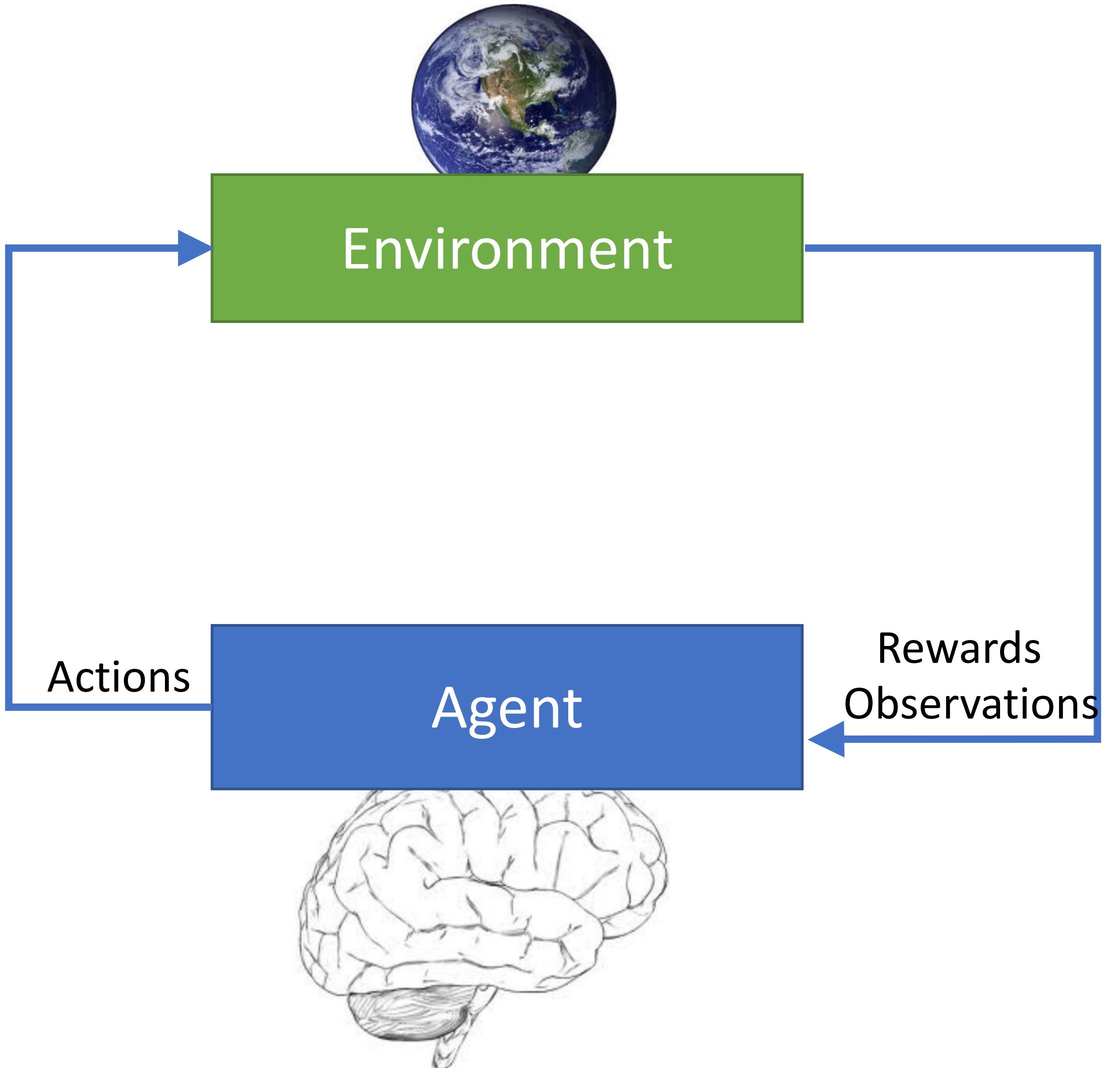
Università degli Studi di Trieste
II Semestre 2020

Lectures 19-20: Reinforcement Learning

What is Reinforcement Learning

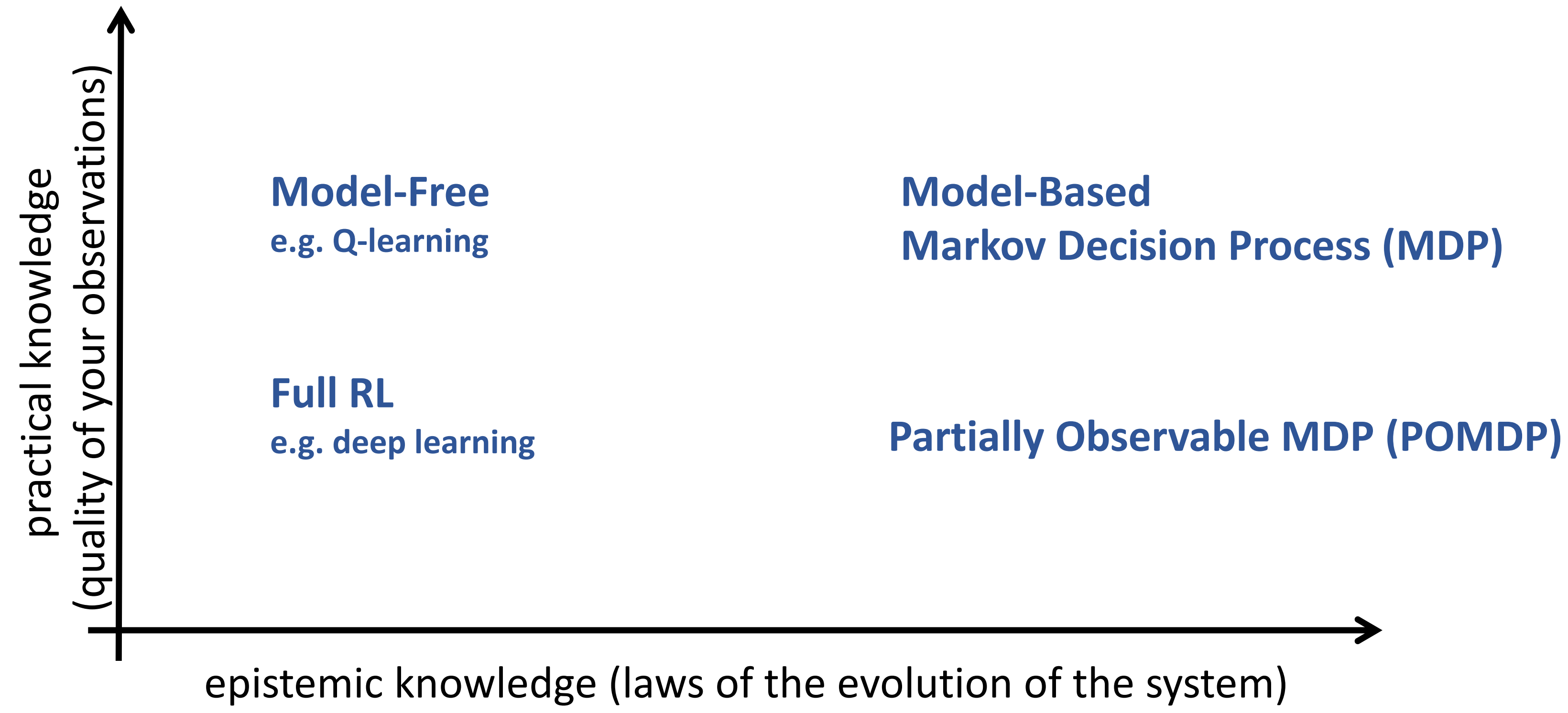


What is Reinforcement Learning



- RL is the theoretical model for learning from interaction with an uncertain environment
- aleatory (intrinsic) or epistemic (knowledge) uncertainty
- **Maximize the average reward function over a given time horizon**
- Very important notion of time horizon, it can change your goal
- There could be different reward to achieve the same goal

What is Reinforcement Learning

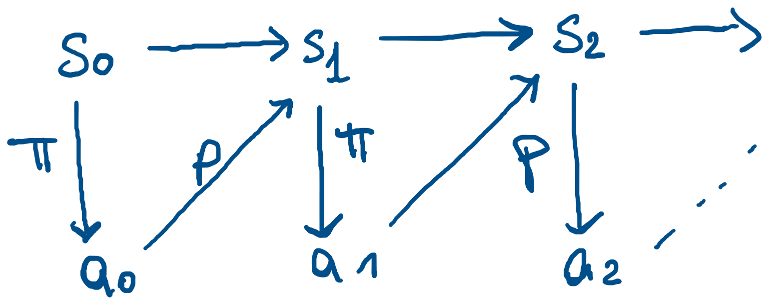


Markov Decision Process

- MDP can be described as a tuple $(S, A, P, \pi, R, \gamma)$, where:
  - S: discrete countable set of states
  - A: set of actions
  - $P: S \times A \times S \rightarrow [0,1]$ is the transition probability function s.t. $P(s, \mathrm{a}, s') = \Pr(s'|s, a)$. It is the model of the environment
  - $\pi{:}S \rightarrow A$ is the policy function mapping states to actions, (Deterministic policy $\mathrm{a} = \pi(s)$, Stochastic policy, $\pi(a|s) = \Pr(\mathrm{a}|\mathrm{s})$)
  - $R: S \times A \times S \rightarrow \mathbb{R}$ is a reward function. We will use only state-reward functions to make it easy ($R: S \rightarrow \mathbb{R}$)
  - $\gamma \in [0,1]$ is a discount factor representing diminishing rewards with time

MDP run

- Start in some initial state s_0 and choose action a_0 with respect π
  - Results in some state s_1 drawn according to $s_1 \sim P(s_0, \mathrm{a}_0)$
    - Pick a new action a_1
      - Results in some state s_2 drawn according to $s_2 \sim P(s_1, \mathrm{a}_1)$
        - ...



- Total payoff for this run:

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \ \ldots$$

MDP as two-player game

- System starts in some initial state s_0 and player 1 (controller) chooses action a_0
  - Results in player 2 (environment) picking state s_1 according to $s_1 \sim P(s_0, \mathrm{a}_0)$
    - Player 1 picks a new action a_1
      - Player 2 picks state s_2 drawn according to $s_2 \sim P(s_1, \mathrm{a}_1)$
        - …
- Total payoff for this run:

$$R(s_0\,) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \ \ldots$$

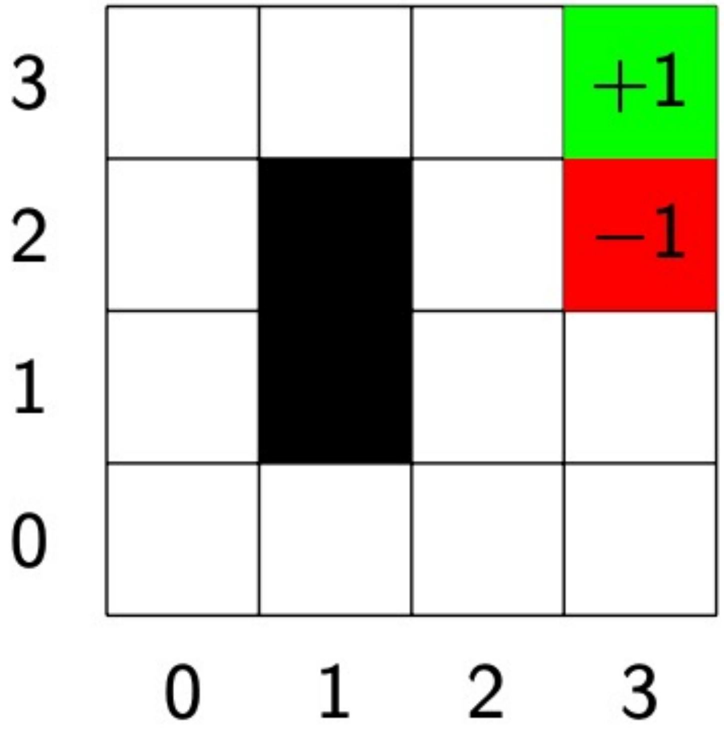
Policies

- Policy π is basically the "implementation" of our controller. It tells the controller what action to take in each state.
- If we are executing policy π, then in state s, we take action an $a = \pi(s)$
- Goal: **Maximize the average reward function over a given time horizon**
- Maximize over $(\pi(s_0), \dots, \pi(s_{T-1}))$ the average reward function

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t, s_t')\right]$$

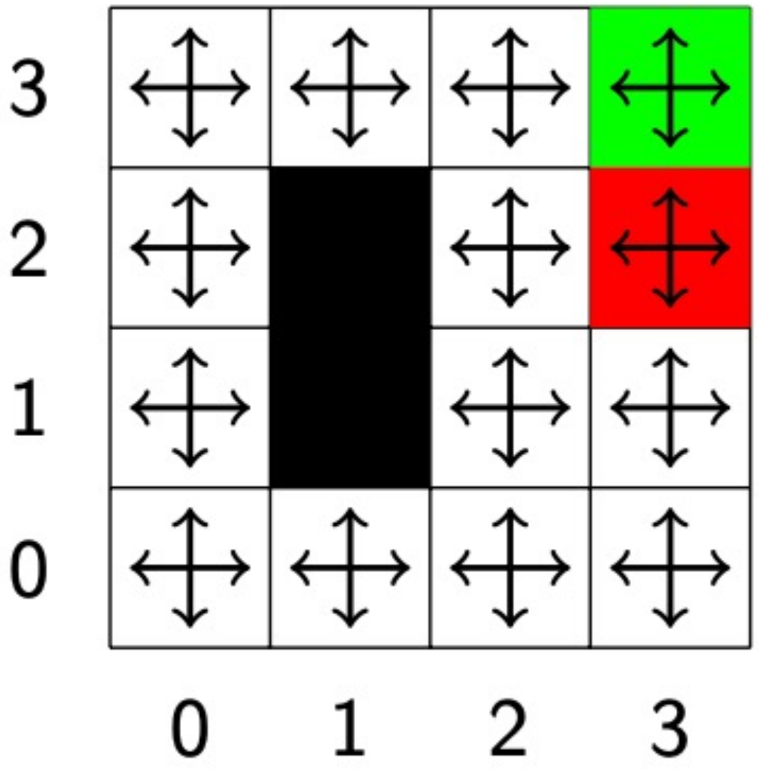
- $\gamma \in [0,1]$ is the discount factor, you can see it as the probability of survival
- $\frac{1}{1-\gamma}$ is the effective **time horizon**

Example: Grigworld



- S = {(i,j)|i,j ∈ [0,3]},i.e., each cell in the grid.
- A = {UP, DOWN, LEFT, RIGHT}
- $P(s'|s,a) =$
$$\begin{cases} 1-\epsilon & \text{if s'= s + a} \\ \frac{\epsilon}{4} & other\ neighbouroad \end{cases}$$
- R: +1 for goal (green) and −1 for fail (red), else 0.

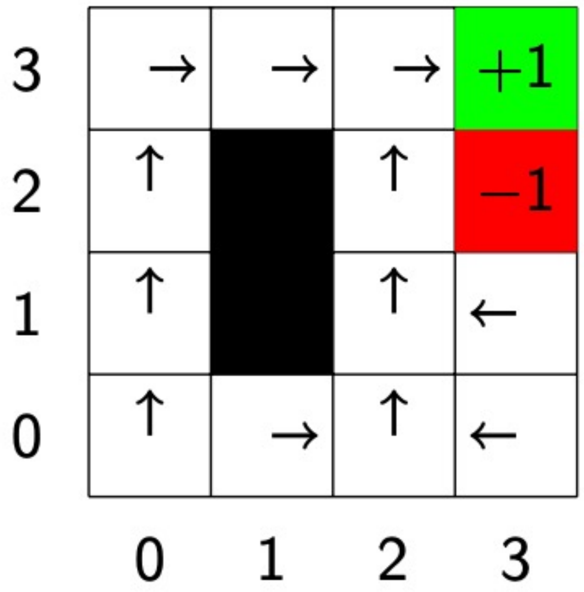
Example: Grigworld



Random Policy π(s, a) = 0.25

A Random agent is one that uniformly picks an action from the action space A.

Example: Grigworld



Policy under deterministic MDP (P(success) = 1)

- In the deterministic MDP case, you can use your favorite path planning algorithm (Dijkstras, Bellman-Ford, .., i.e. algorithm to compute the shortest path) to find a the optimal policy.
- We learn a policy π such that $\pi(s, a) = 1$ for correct action, 0 otherwise.

Value Function

- Value function of a state s under policy π (denoted $V_t^{\pi}(s)$) is a prediction of future reward, "How much reward will I get from action a in state s?"
- I.e. $V_t^{\pi}(s) = \mathbb{E}\left[\sum_{t'=t}^{T-1} \gamma^{t'} R(s_{t'}) \mid s_t = s\right]$

$$V_t^{\pi}(s) = \sum_{s'} P(s,a,s')\,[R(s) + \gamma V_{t+1}^{\pi}(s')]$$

$$= R(s) + \gamma \sum_{s'} [P(s,a,s')\, V_{t+1}^{\pi}(s')]$$

Computing optimal reward/cost over several steps of a dynamic discrete decision problem (i.e. computing the best decision in each discrete step) can be stated in a recursive step-by-step form by writing the relationship between the value functions in two successive iterations.

Bellman's Equation

- $V_t^\pi(s) = R(s) + \gamma \sum_{s'} [P(s, a, s')\, V_{t+1}^\pi(s')]$

- I.e. expected sum of rewards starting from s has two terms:
  - Immediate reward $R(s)$
  - Expected sum of future discounted rewards

- Note that above is the same as:

$$V_t^\pi(s) = R(s) + \mathbb{E}_{s' \sim P(s, \pi(s), s')}[V_{t+1}^\pi(s')]$$

Bellman's Equation time-independent

- $V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t'} R(s_t) \mid s_0 = s\right]$
- $V^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s, \pi(s), s') V^{\pi}(s')$
- For a finite-state MDP, we can write one such equation for each s, which gives us $|S|$ linear equations in $|S|$ variables (the unknown $V^{\pi}(s)$ for each s).

Optimal value function

- We now know how to compute the value for a given policy
- Computing best/optimal policy:

$$V_*(s) = \max_{\pi} V^{\pi}(s)$$

- There is a Bellman equation for optimal value function:

$$V_*(s) = R(s) + \gamma \max_{\mathrm{a} \in A} \sum_{s' \in S} P(s, \mathrm{a}, s') V_*(s')$$

- And optimal policy is the a$'s$ that make above equation hold, i.e.

$$\pi^*(s) = \mathrm{argmax}_{\mathrm{a} \in \mathrm{A}} \; P(s, \mathrm{a}, s') V_*(s')$$

Planning in MDPs

- How do we compute the optimal policy?
- Two algorithms:
  - Value iteration
  - Policy iteration

- Value iteration: Repeatedly update estimated value function using Bellman equation
- Policy iteration: Use value function of a given policy to improve the policy

Value iteration

$V_k(s)$: Value of state s at the beginning of the k^{th}iteration

Initialize $V(s) \coloneqq 0, \forall s \in S$

While $\left(\max_{s \in S} |V_{k+1}(s) - V_k(s)|\right) \geq \epsilon$ {

$$V_{k+1}(s) \coloneqq R(s) + \gamma \max_{\mathrm{a} \in A} \left\{ \sum_{s'} P(s, \mathrm{a}, s') V_k(s') \right\}$$

- Can be shown that after finite number of iterations V converges to V_*

Policy iteration

Let π_k be the policy at the beginning of the k^{th} iteration

Initialize π randomly
While $(\exists s :\ \pi_{k+1}(s) \neq \pi_k(s))$ {
$V := V^{\pi}$ /* i.e. $\forall s$ compute $V^{\pi}(s)$ */
$\pi_{k+1}(s) := \arg\max_{a \in A} \sum_{s'} P(s, a, s')\, V(s')$
}

Can use the LP formulation to solve this, or an iterative algorithm

- Can be shown that this algorithm also converges to the optimal policy

Using state-action pairs for rewards

$$Q^{\pi}(s,a) = \sum_{s'} P(s,\pi(s),s') \left[R(s,a,s') + \sum_{a'} \pi(a'|s')\, Q^{\pi}(s',a') \right]$$

- $Q^{\pi}(s,\mathrm{a})$ is called the Quality function or Stat-Action Value function and indicates the reward obtained by taking action a in state s and following the policy π thereafter

- Optimal-action-value policy denoted by Q_*

$$Q_*(s,a) = \sum_{s'} P(s,\mathrm{a},s')(R(s,\mathrm{a},s') + \gamma \max_{a\prime}(Q_*(s',a'))$$

- Note that previous formulas change a bit, as the reward depends on which action is taken (and is thus is subject to transition probability)

Challenges

- Value-fcn requires less memory
- Q-fcn makes the choice of optimal action more straightforward
- Value iteration is preferred over policy iteration as the latter requires solving linear equations, which scales ~cubically with the size of the state space

- Real-world applications face challenges:
  1. Curse of modeling: Where does the (probabilistic) environment model come from?
  2. Curse of dimensionality: Even if you have a model, computing and storing expectations over large state-spaces is impractical. -> functional approximation

Example: Value iteration Gridworld

3	0	0	0	+1
2	0		0	−1
1	0		0	0
0	0	0	0	0
	0	1	2	3

Value after one iteration

3	0	0	0.72	+1
2	0		0	−1
1	0		0	0
0	0	0	0	0
	0	1	2	3

Value after 2 iterations

Example: Value iteration Gridworld

3	0	0.52	0.78	+1
2	0		0.43	−1
1	0		0	0
0	0	0	0	0
	0	1	2	3

Value after 3 iterations

3	0.37	0.66	0.83	+1
2	0		0.51	−1
1	0		0.31	0
0	0	0	0	0
	0	1	2	3

Model-based method

The agent is assumed to have prior knowledge about the effects of its actions on the environment, that is, the transition probability function P of the MDP is known.

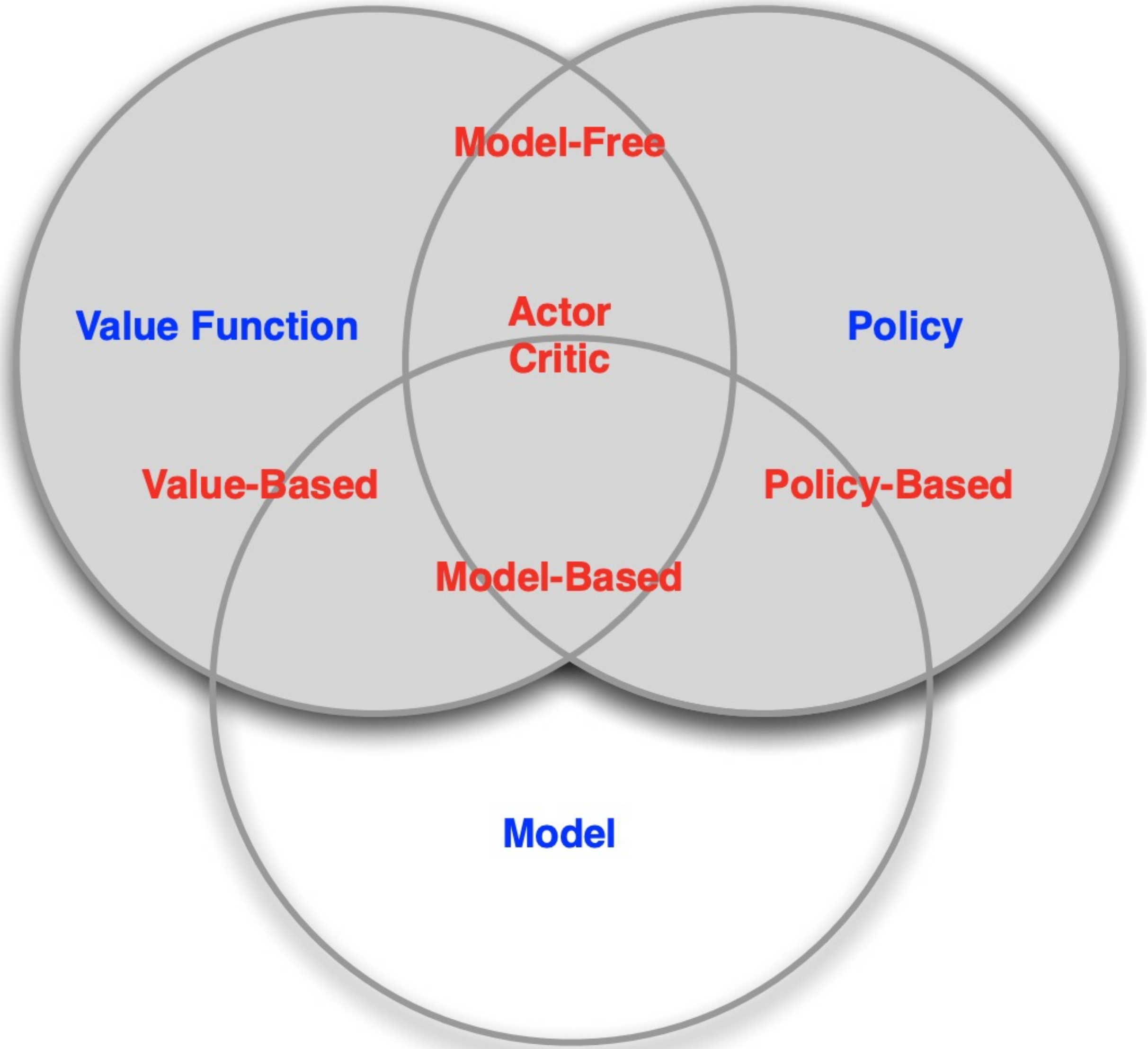
Policy iteration and Value iteration are model-based methods as it is necessary to have knowledge of the probability of transitions in the MDP to compute the expected V (s) at any given iteration of the algorithm.

Planning by dynamic programming, solving a known MDP

Model-free

- Called a model-free method, because it does not assume knowledge of a model of the environment
- Learning agent tries to learn optimal policy from its history of interactions with the environment

General Picture



Model-free methods

Model-free prediction

Estimate the value function of an unknown MDP

- Monte-Carlo Learning
- Temporal-Difference Learning
- TD(λ)

Model-free control

Optimize the value function of an unknown MDP

Uses of Model-Free Control

Some example problems that can be modelled as MDPs

Elevator, Parallel Parking, Ship Steering, Bioreactor, Helicopter, Aeroplane Logistics, Soccer, Quake, Portfolio management, Protein Folding, Robot walking, Game of Go

For most of these problems, either:

- MDP model is unknown, but experience can be sampled
- MDP model is known, but is too big to use, except by samples

Model-free control can solve these problems

Monte-Carlo Reinforcement Learning

- MC methods learn directly from episodes of experience
- MC is model-free: no knowledge of MDP transitions / rewards
- MC learns from **complete episodes**: no bootstrapping
- MC uses the simplest possible idea: value = mean return
- Caveat: can only apply MC to episodic MDPs:
  - all episodes must terminate

Monte-Carlo Reinforcement Learning

- Monte-Carlo policy evaluation uses empirical mean return instead of expected return

- Recall that the **return** is the total discounted reward:

$$G_t \;=\; R_{t+1} + \gamma R_t + 2 \;+\; \ldots + \gamma^{T-1}\; R_T$$

- Recall that the value function is the expected return:

$$V^{\pi}{}_t(s) = \mathbb{E}\left[\sum_{t'=t}^{T-1} \gamma^{t'} R(s_{t'}) \,|\, s_t = s\right] = E_{\pi}\,[G_t|s_t \;=\; s]$$

First-Visit Monte-Carlo Policy Evaluation

- To evaluate state s
- The first (every) time-step t that state s is visited in an episode,
- Increment counter $N(s) \leftarrow N(s) + 1$
- Increment total return $S(s) \leftarrow S(s) + G_t$
- Value is estimated by mean return $V(s) = S(s)/N(s)$
- By law of large numbers, $V(s) \rightarrow V_\pi(s)\ as\ N(s) \rightarrow \infty$

Incremental Mean

The mean of a sequence can be computed incrementally:

$$
\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}\left(x_k + (k-1)\mu_{k-1}\right) \\
&= \mu_{k-1} + \frac{1}{k}\left(x_k - \mu_{k-1}\right)
\end{aligned}
$$

Incremental Monte-Carlo

Update V(s) incrementally after episode $S_1, A_1, R_2, \dots, S_T$

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

In non-stationary problems, it can be useful to track a running mean, i.e. forget old episodes

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Temporal-Difference Learning

- TD methods learn directly from episodes of experience
- TD is model-free: no knowledge of MDP transitions / rewards
- TD learns from **incomplete** episodes, by **bootstrapping**
- TD updates a guess towards a guess

MC and TD

- Goal: learn V_π online from experience under policy π
- Incremental every-visit Monte-Carlo Update value $V(S_t)$ toward actual return G_t

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$

- Simplest temporal-difference learning algorithm: TD(0)
  - Update value $V(S_t)$ toward estimated return: $R_{t+1} + \gamma V(S_{t+1})$

$$V(S_t) \leftarrow V(S_t) + \alpha\,\underbrace{\Big(\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{this is called the }\textbf{TD target}} - V(S_t)\Big)}_{\text{is called the }\textbf{TD error}}$$

Advantages and Disadvantages of MC vs. TD

MC

- must wait until end of episode before return is known
- can only learn from complete sequences and/or in terminating environments
- has high variance, zero bias

- Good convergence properties (even with function approximation)
- Not very sensitive to initial value
- Very simple to understand and use
- Does not exploit Markov property Usually more effective in non-Markov environments

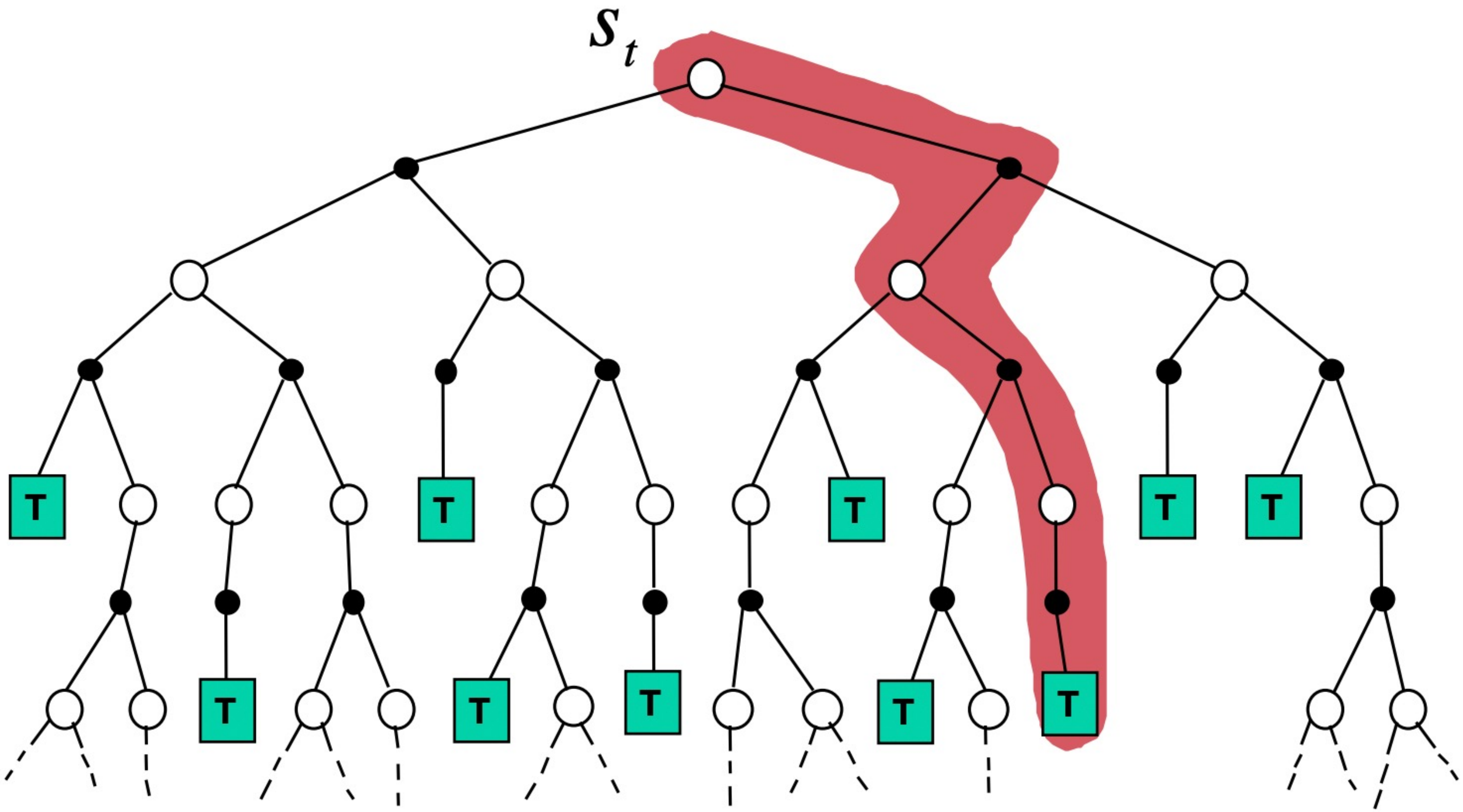
TD

- can learn before knowing the final outcome, learning online after every step
- can learn without the final outcome, from incomplete sequences and/or in non-terminating environments
- has low variance, some bias

- converges to $v_\pi(s)$ (but not always with function approximation)
- More sensitive to initial value
- Usually more efficient than MC TD(0)
- Exploits Markov property (usually more efficient in Markov environments)

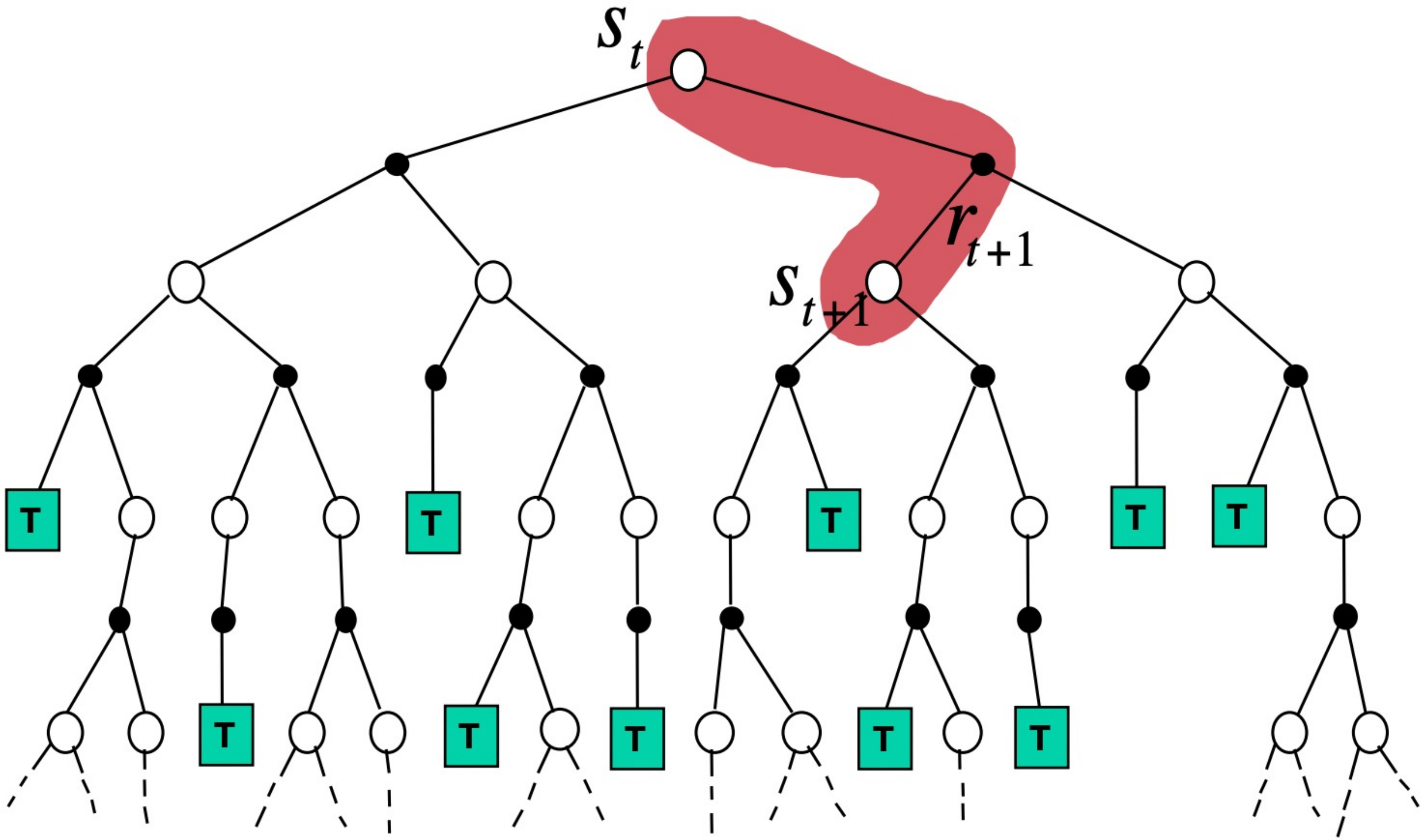
Monte-Carlo Backup

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$



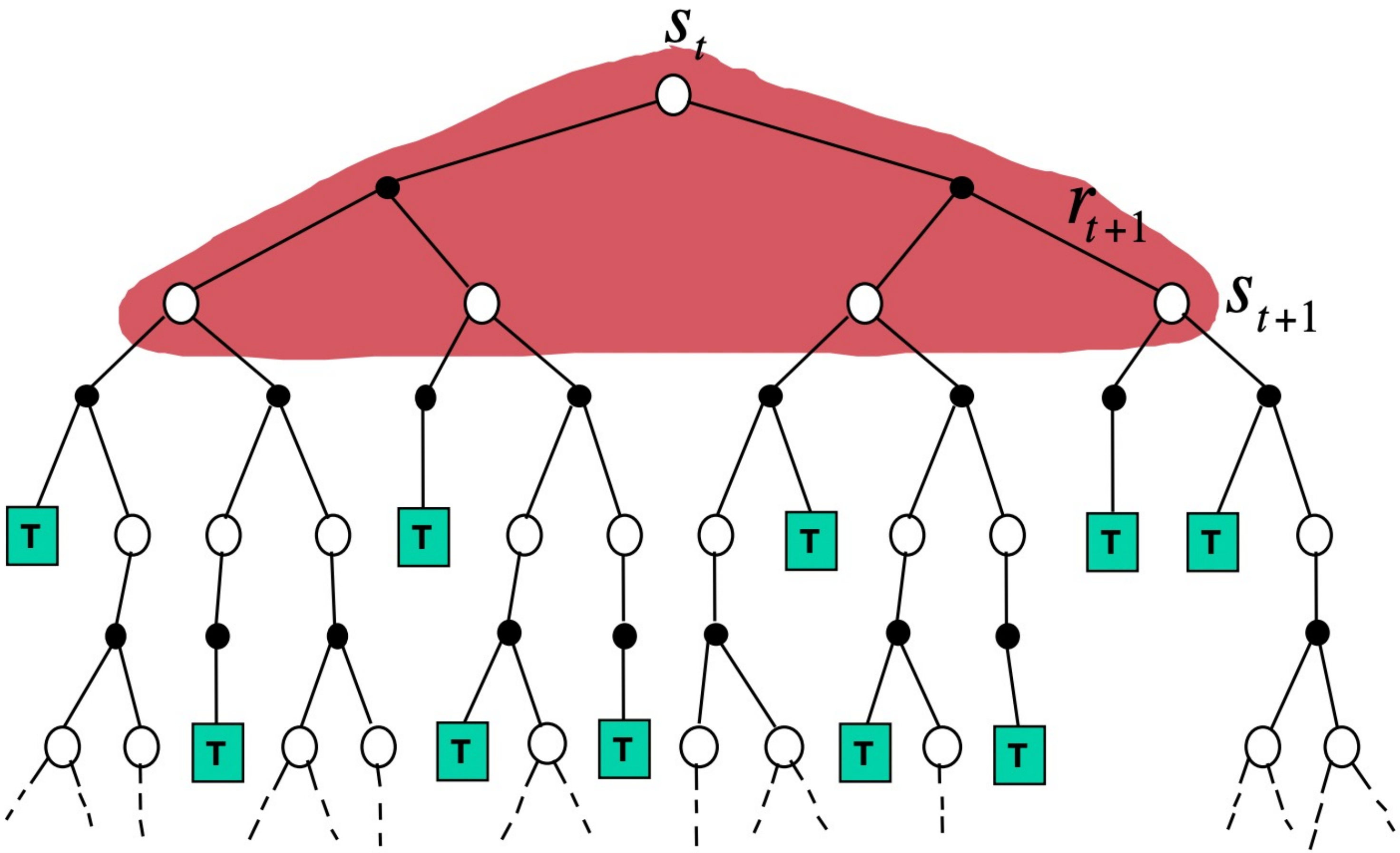
Temporal-Difference Backup

$$V(S_t) \leftarrow V(S_t) + \alpha \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right)$$

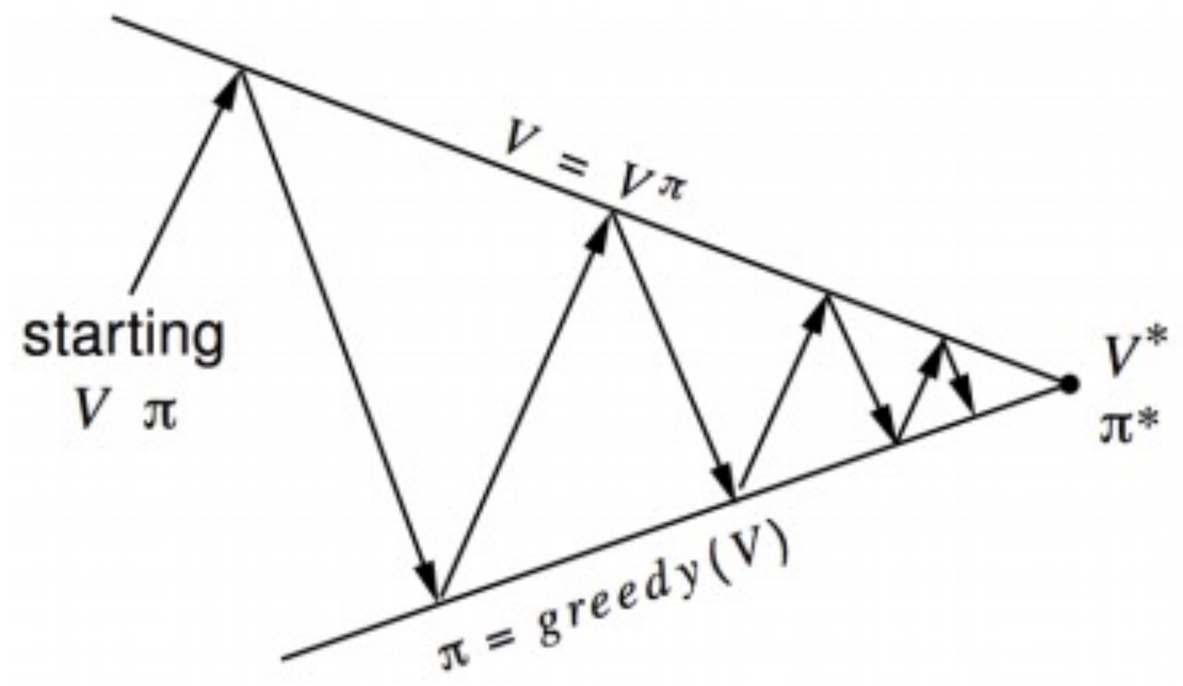


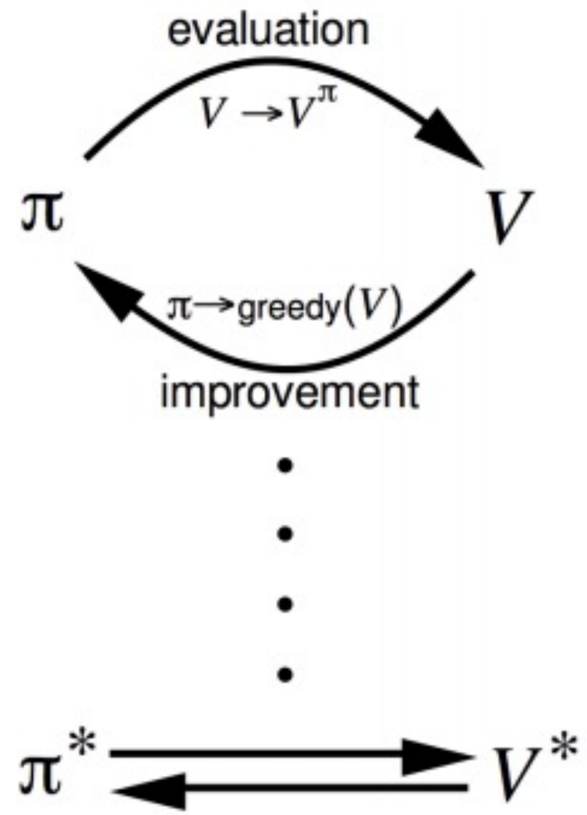
Dynamic Programming Backup

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1})\right]$$



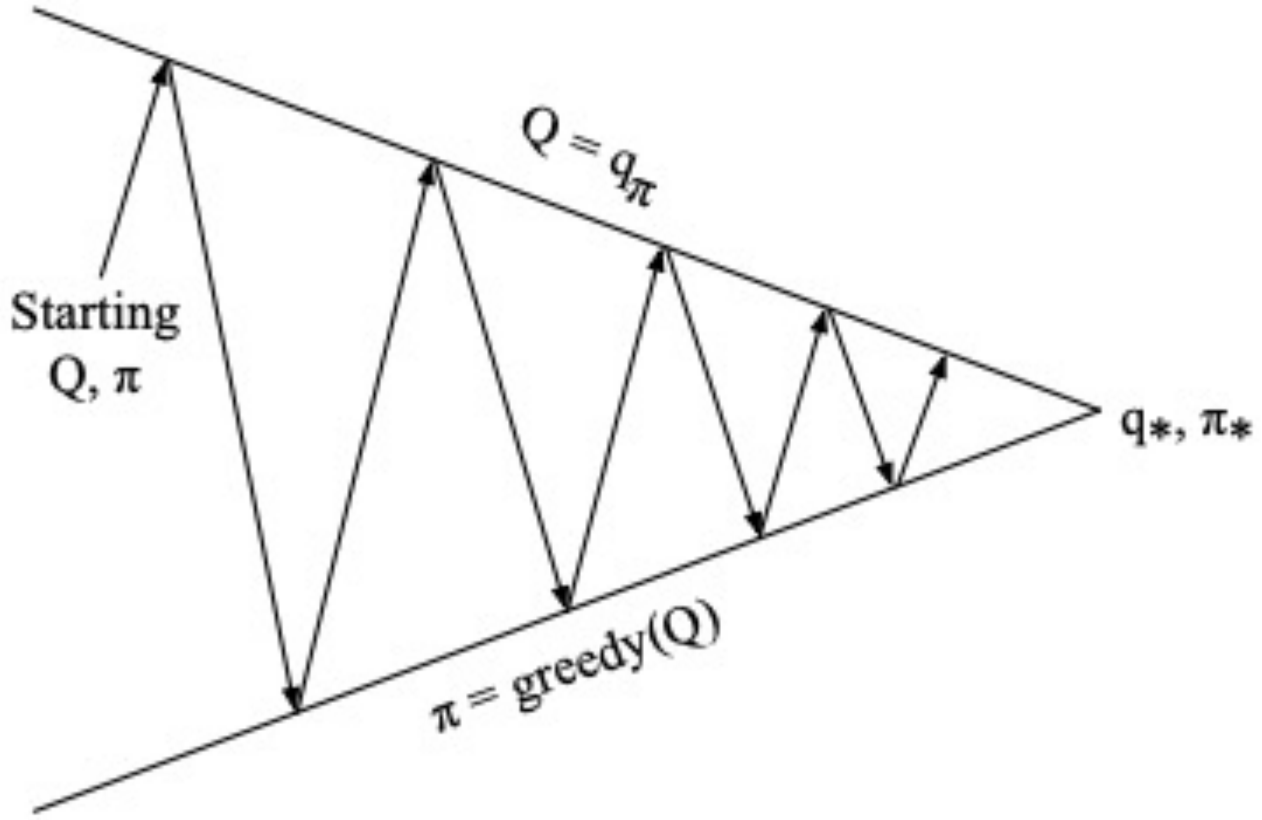
Generalised Policy Iteration (Refresher)





Policy evaluation: Estimate V_π e.g. Iterative policy evaluation
Policy improvement: Generate $\pi' \geq \pi$ e.g. Greedy policy improvements

Generalised Policy With Monte-Carlo Evaluation



Policy evaluation: Monte-Carlo policy evaluation, $Q = q_\pi$
Policy improvement: Greedy policy improvement

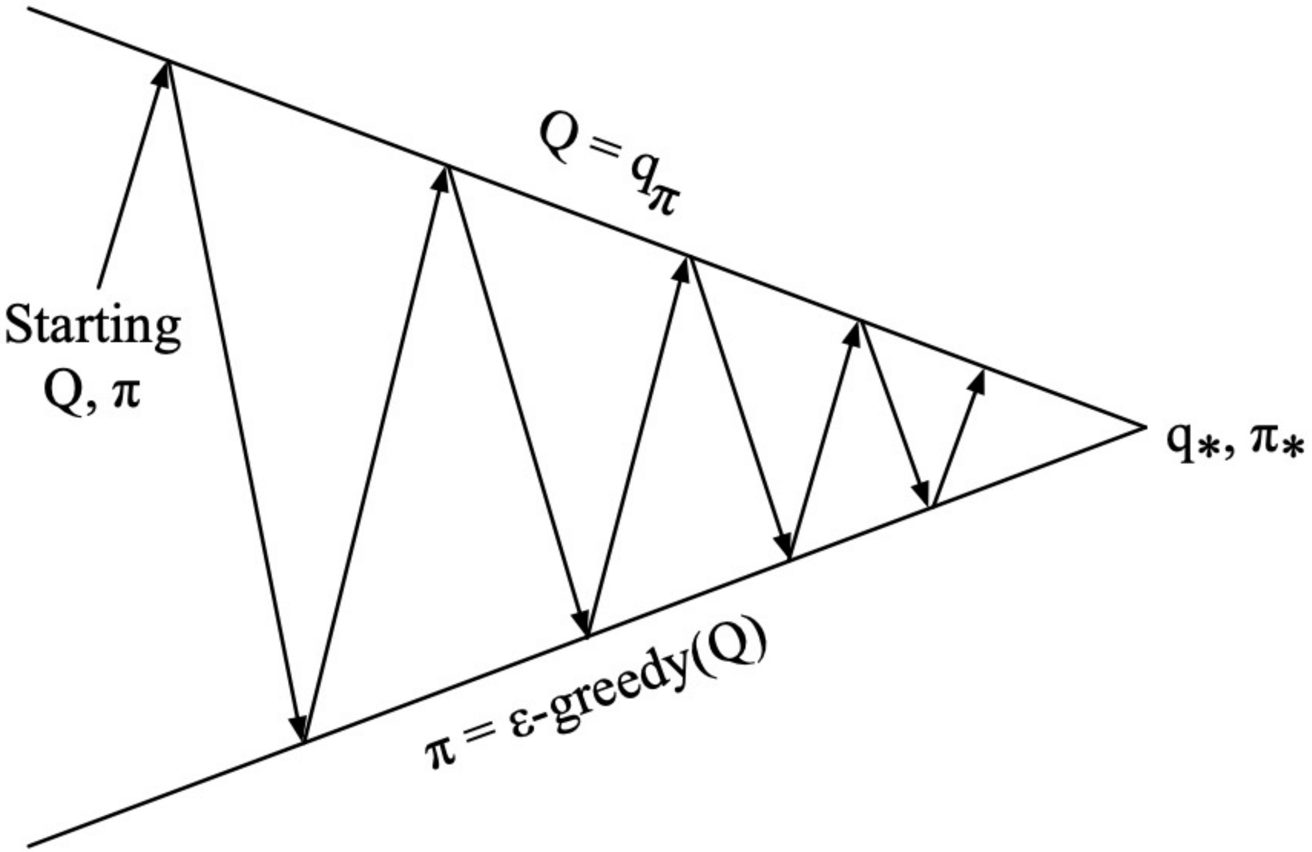
ϵ-Greedy Exploration

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability 1 – ϵ choose the greedy action
- With probability ϵ choose an action at random

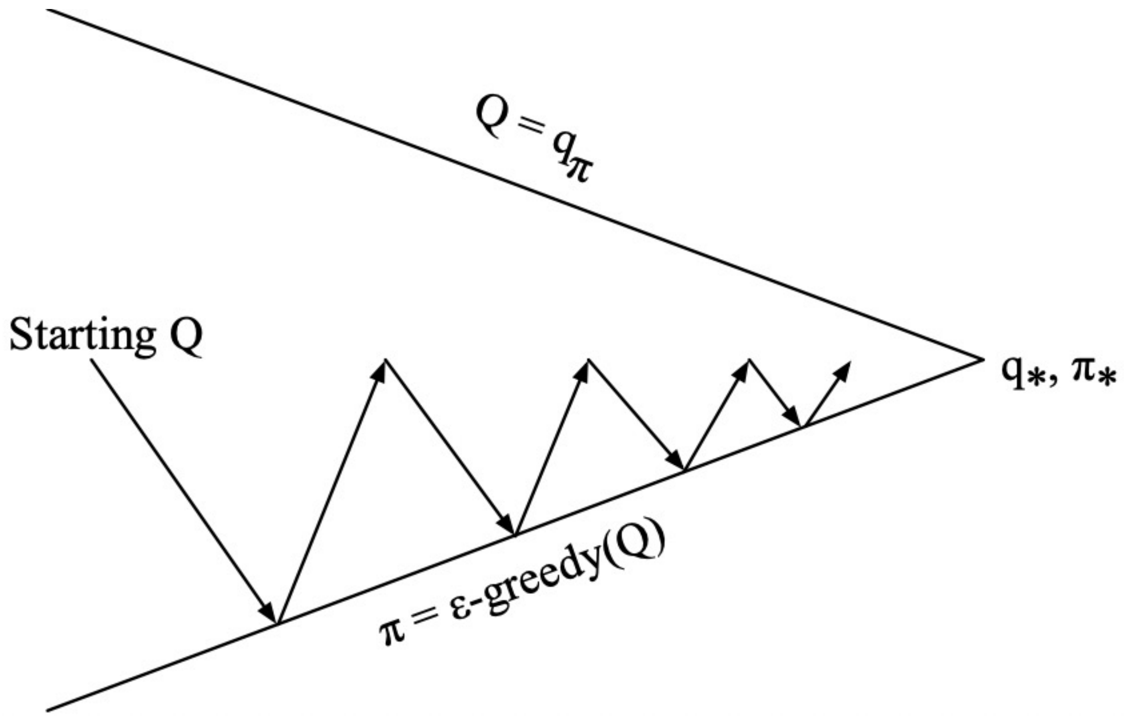
For any ϵ-**greedy policy** π, the ϵ-**greedy** policy π' with respect to q_π is an improvement, i.e. $V_{\pi'}(s) \geq V_\pi(s)$

Monte-Carlo Control



Policy evaluation: Monte-Carlo policy evaluation, $Q\ =\ q_\pi$
Policy improvement: ϵ-Greedy policy improvement

Monte-Carlo Control



Every episode:

Policy evaluation: Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement: ϵ-Greedy policy improvement

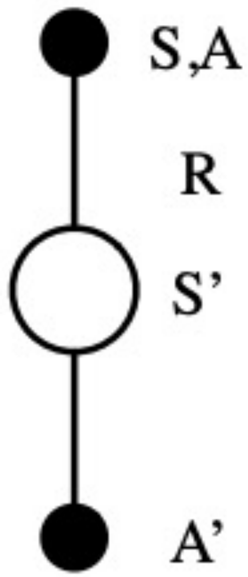
MC vs. TD

TD advantages:

- Lower variance
- Online
- Incomplete sequences

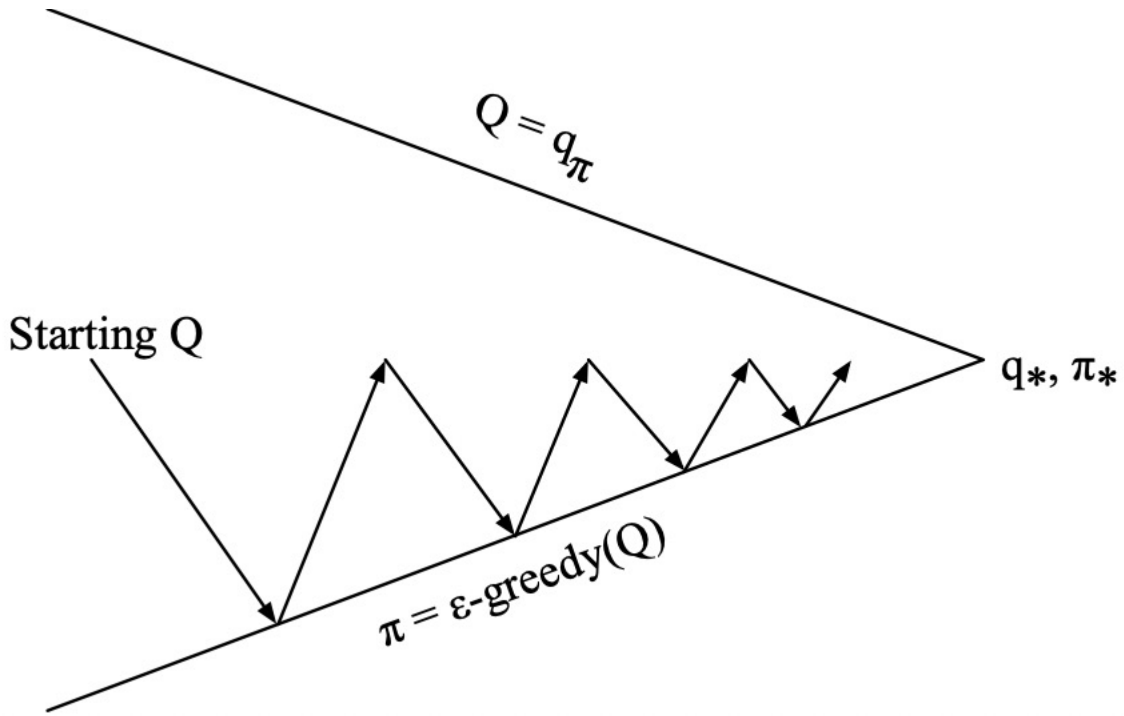
- Natural idea: use TD instead of MC in our control loop:
  - Apply TD to Q(S, A)
  - Use ϵ-greedy policy improvement
  - Update every time-step

SARSA



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma Q(S',A') - Q(S,A) \right)$$

On-Policy With Sarsa



Every time-step:

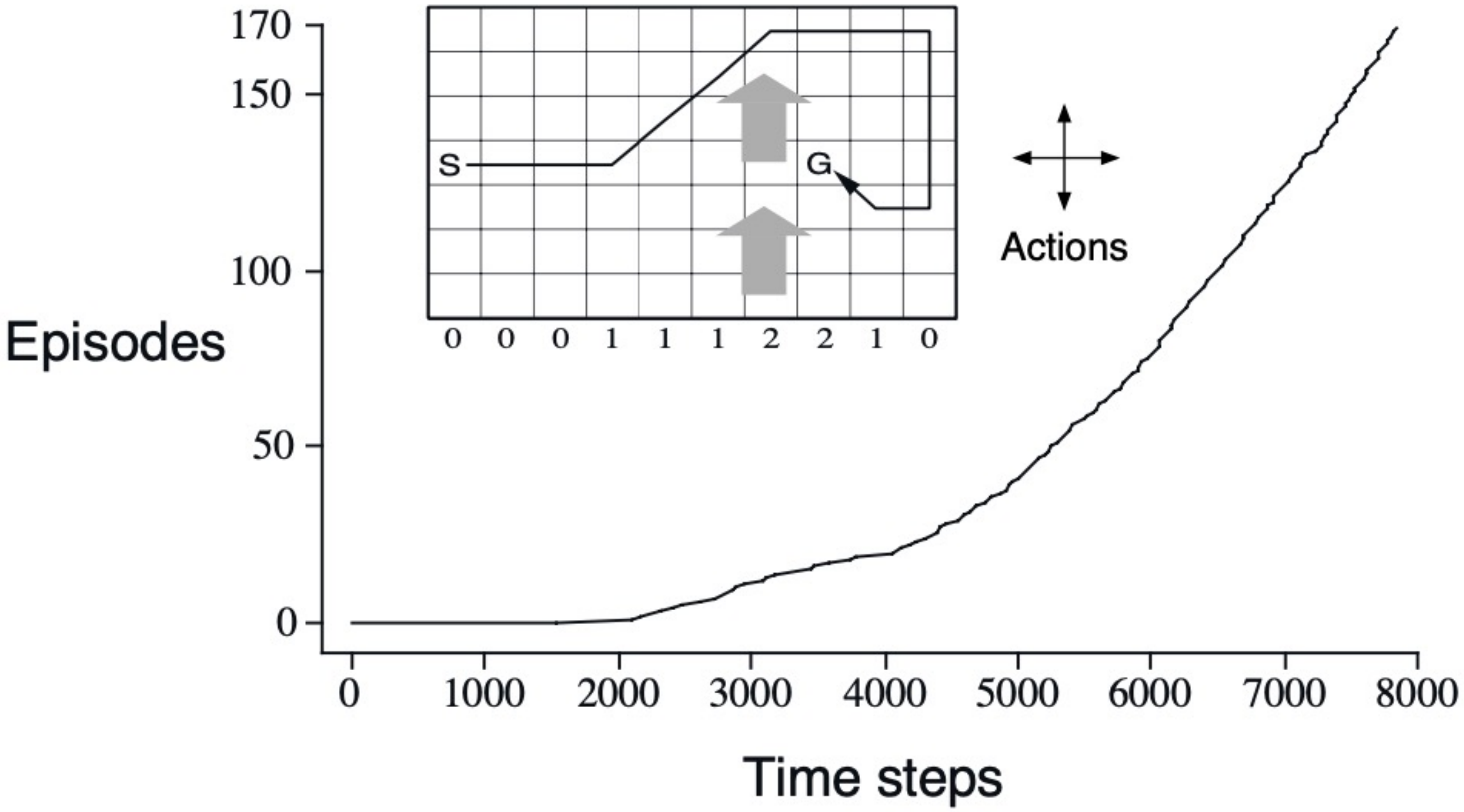
Policy evaluation: Sarsa , $Q \approx q_\pi$

Policy improvement: ϵ-Greedy policy improvement

On-Policy With Sarsa

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Example windy Gridworld



- Reward = -1 per time-step until reaching goal

Off-policy Learning

- Evaluate **target** policy π(a|s) to compute $v_\pi(s)$ or $q_\pi(s,a)$
- While following **behaviour** policy $\mu(a|s)$

$$\{S_1, A_1, R_2, \ldots, S_T\} \sim \mu$$

- Why is this important?
  - Learn from observing humans or other agents
  - Re-use experience generated from old policies $\pi_1, \pi_2, \ldots, \pi_{t-1}$
  - Learn about optimal policy while following exploratory policy
  - Learn about multiple policies while following one policy

Q-learning: Off-policy TD Control

- We now consider off-policy learning of action-values $Q(s, a)$
- Next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot\, | S_t)$
- But we consider alternative successor action $A' \sim \pi(\cdot\, | S_t)$
- And update Q(St , At) towards value of alternative action

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t) \right)$$

Q-learning: Off-policy TD Control

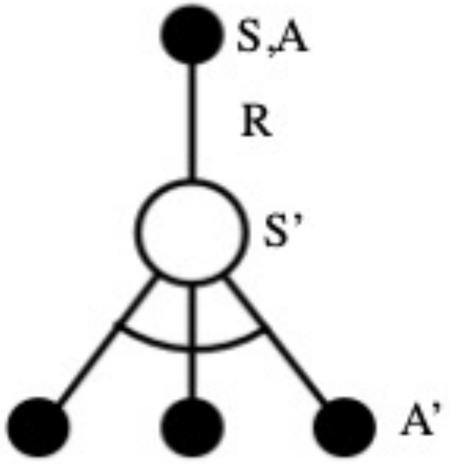
- We now allow both behaviour and target policies to improve
- The target policy π is greedy w.r.t. Q(s, a)

$$\pi(S_{t+1}) = argmax_{a'} Q(S_{t+1}, a')$$

- The behaviour policy μ is e.g. ϵ-greedy w.r.t. Q(s, a)
- The Q-learning target then simplifies:

$$R_{t+1} + \gamma Q(S_{t+1}, a') = R_{t+1} + \gamma Q(S_{t+1}, argmax_{a'} Q(S_{t+1}, a')) =$$
$$= R_{t+1} + \max_{a} \gamma Q(S_{t+1}, a')$$

Q-learning: Off-policy TD Control



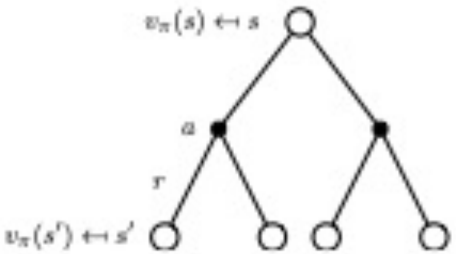
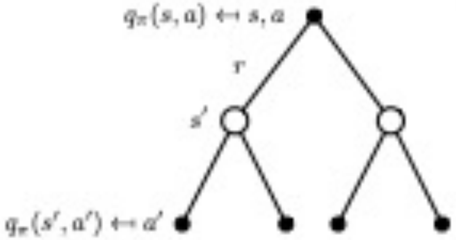
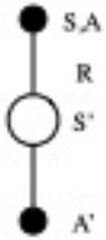
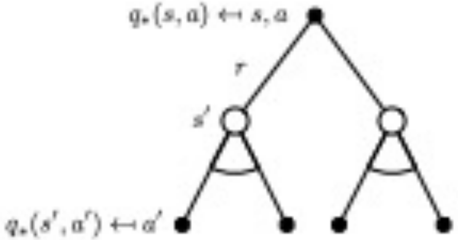
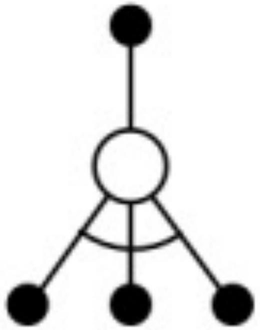
$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Q-learning

- Whenever the agent is in state q and takes action a, we have new data about the reward that we get, we use this to update our estimate of the Q value at that state
- Agent updates its estimate of $Q(s,\mathrm{a})$ using following equation:

$$Q(s,\mathrm{a}) \coloneqq Q(s,\mathrm{a}) + \alpha\left(r + \gamma \max_{\mathrm{a}'\in A}(Q(s',\mathrm{a}')\ - Q(s,\mathrm{a}))\ \right)$$
$$\coloneqq (1-\alpha)Q(s,\mathrm{a}) + \alpha\left(r + \gamma \max_{\mathrm{a}'} Q(s',\mathrm{a}')\right)$$

- Learning rate α controls how aggressively you update the old Q value.
  - $\alpha \approx 0$ means that you update Q value very slowly
  - $\alpha \approx 1$ means that you simple replace the old value with the new value
- $\max_{a\prime} Q(s',\mathrm{a}')$ is the estimate of the optimal future value

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 Q-Value Iteration	 Q-Learning

Bibliography

This is a subset of the sources I used. It is possible I missed something!

1. Richard S. Sutton and Andrew G. Barto, Reinforcement Learning, MIT Press.
2. http://ieeecss.org/CSM/library/1992/april1992/w01-ReinforcementLearning.pdf
3. https://github.com/omerbsezer/Reinforcement_learning_tutorial_with_demo - FunctionApproximation